CMPSCI 377 Operating Systems **Fall 2013**

Lecture 7: September 25

Lecturer: Prashant Shenoy *Scribe: Armand Halbert*

7.1 Synchronization

As we already know, threads must ensure consistency; otherwise, race conditions (non-deterministic results) might happen.

Now consider the "too much milk problem": two people share the same fridge and must guarantee that there's always milk, but not too much milk. If the two people do not coordinate, it is possible that both will go to buy milk at once and have too much milk. How can they solve this? First, we consider some important concepts and their definitions:

- **Synchronization**: the use of atomic operations to allow coordination between threads (same ideas work for processes also);
- **Mutual Exclusion**: ensuring that only one thread is performing a particular action or accessing a piece of data at a time (and excluding other threads);
- **Critical section**: a piece of code that only one thread can execute at a time;
- **Lock**: a mechanism for mutual exclusion; the program locks on entering a critical section, accesses the shared data, and then unlocks. Any other programs must wait if they try to enter a locked section.

For the above mentioned problem, we want to ensure some correctness properties. First, we want to guarantee that only one person buys milk when it is needed (this is the *safety* property, aka "nothing bad happens"). Also, we want to ensure the *liveness* property – that someone *does* buy milk when needed. Now consider that we can use the following atomic operations when writing the code for the problem:

- "leave a note" (equivalent to a lock)
- "remove a note" (equivalent to a unlock)
- "don't buy milk if there's a note" (equivalent to a wait)

Our first try could be to use the following code on both threads:

```
if (no milk and no note)
   leave note
   buy milk
   remove note
```

Unfortunately, this doesn't work because both threads could simultaneously verify that there's no note and no milk, and then both would simultaneously leave a note, and buy more milk. The problem in this case is that we end up with too much milk (safety property not met).

Now consider our solution #2 using labelled notes:

Thread A:

```
leave note "A"
if (no note "B")
   if (no milk)
      buy milk
remove note "A"
```

Thread B:

```
leave note "B"
if (no note "A")
   if (no milk)
      buy milk
remove note "B"
```

The problem now is that if both threads leave notes at the same time, neither will ever do anything. Then, we end up with no milk at all, which means that the liveness property is not met.

7.2 Concurrency

When programming with threads, processes or with any type of program that has to deal with shared data, we have to take into account all possible interleaving of these processes. In other words, in order to guarantee that concurrent processes are *correct*, we have to somehow guarantee that they generate the correct solution no matter how they are interleaved.

¿From the "Too Much Milk" problem it is clear that it can be very difficult to come up with an approach that always solves it properly. Let us now consider an approach that *does* work:

Thread A

```
leave note A
while (note B)
   do nothing
if (no milk)
   buy milk
remove note A
```

Thread B

```
leave note B
if (no note A)
   if (no milk)
      buy milk
remove note B
```

This approach, unlike the two previous examples, does work. However, it is not easy to be convinced that these two algorithms, when taken together, always produce the desired behavior. Moreover, these pieces of code have some drawbacks: first, notice that Thread A goes into a loop waiting for B to release its note. This is called "busy waiting", and is generally not a good idea because Thread A wastes a lot of CPU, and because it can't execute anything useful while B is not done. Also, notice that even though both threads try to perform the exact same thing, they do it in very different ways. This is a problem specially when we were to write, say, a third thread. This third thread would probably look very different than both A and B, and this type of asymmetric code does not scale very well. So the question is: how can we guarantee correctness and at the same time avoid all these drawbacks? The answer is that we can augment the programming language with high-level constructs capable of solving synchronization problems. Currently, the best known constructs used in order to deal with concurrency problems are *locks, semaphores, monitors.*

7.2.1 Locks/Mutex

Locks (also known as Mutexes) provide mutual exclusion to shared data inside a critical section. They are implemented by means of two atomic routines: *acquire*, which waits for a lock, and takes it when possible; and *release*, which unlocks the lock and wakes up the waiters. The rules for using locks/mutex are the following:

1. only one person can have the lock at a time;
2. locks must be acquired before accessing shared data;
3. locks must be released after use;
4. locks are initially released.

Let us now try to rewrite the "Too Much Milk" problem in a cleaner and more symmetric way, using locks. In order to do so, the code for Thread A (and also for Thread B) has to be the following:

```
lock.acquire()
if (no milk)
   buy milk
lock.release()
```

This is clearly much easier to understand than the previous solutions; also, it is more scalable, since all threads are implemented in the exact same way.

7.3 Implementing Locks

To implement a lock, the OS needs a way to ensure that a process will be able to run a *critical section* where it can access some shared data without another process modifying the data at the same time. No matter how an OS chooses to implement locks, it *must* have some hardware support.

One way to implement locks is to *disable interrupts*, since interrupts are the only way the OS has to change what it is doing. Normally, a process in an operating system will continue running unless it either performs an I/O request or it is interrupted by the operating system through the use of an interrupt–for example because it's scheduling time quantum has run out or due to some kind of exception. By disabling interrupts,

we can ensure that a process will maintain control of the CPU and guarantee that only one process (the active one) will have access to the shared data. Disabling interrupts will prevent any *external events* from causing the process to lose control of the CPU. In addition, the process must prevent *internal events* as well; this is typically done by not initiating an I/O request while in a critical section.

Another option for implementing locks would be to make use of *atomic operations*, such as test&set. This operation (which usually corresponds to an assembly instruction), is such that test&set(x) returns 1, if x=1; otherwise, if x=0, it returns 0 and sets x to 1. The test$set operation is called an atomic read-modify-write instruction because it reads a value (which is returned) and writes a new value without being able to be interrupted by the OS scheduler. Each of these operations must be implemented atomically by the hardware. Having this type of atomic operation, one could implement $acquire(L)$ simply as

```
while test&set(L) do nothing;
```

and $release(L)$ simply as

```
l = 0;
```

A lock implementation can either use busy waiting or wait queues. In *busy waiting*, when a thread calls acquire(), it will continuously check the lock to see when it becomes available again. While this is simple to implement, it can be inefficient because the thread is using CPU time to constantly check a lock which another thread has control of. An alternative is to use *wait queues*. In this case, when a thread calls acquire(), if some other thread has the lock, then it is placed on a wait queue. This is a list of threads which are all waiting for the lock to become available. These threads will not be scheduled until the thread currently using the lock calls release() and they are woken up. This is a more efficient approach, that tries to minimize busy waiting to inside the acquire() function (which will execute quickly).